C# 프로그래밍

- 13 주차 (2강)

SEJONG UNIVERSITY

일반화 프로그래밍
- 일반화 메소드/클래스
- 형식 매개변수의 제약
- 일반화 컬렉션

일반화 프로그래밍 (Generic Programming)

- 클래스 또는 메소드를 정의 할 때, 내부에 사용되는 데이터형식에 식별자를 지정
  - ✓ 보통 식별자로 "T" 를 많이 사용
  - ✓ 식별자에 원하는 데이터형식을 할당하여 해당 클래스 또는 메소드 사용
- 매개변수의 데이터형식이 다른 오버로딩 메소드들을 효과적으로 일반화
  - ✓ 예를 들어, int 형 배열을 복사하는 메소드(왼쪽)와 string 형 배열을 복사하는 메소드(오른쪽)
  - ✓ Int 형과 string 형을 포함해 31가지 데이터형식으로 오버로딩 메소드 작성해야 할 수도 있음
  - ✓ 일반화 프로그래밍은 데이터형식을 일반화하여 효율적으로 메소드 정의

int 형 배열 복사 메소드:

```
void CopyArray (int[] source, int[] target)
{
    for (int i = 0; i < source.Length; i++)
        target[i] = source[i];
}
```

string 형 배열 복사 메소드:

```
void CopyArray(string[] source, string[] target)
{
    for (int i = 0; i < source.Length; i++)
        target[i] = source[i];
}
```

일반화 메소드

- 매개변수의 데이터형식을 일반화하여 정의한 메소드
  - ✓ 메소드 이름 뒤에 꺽은 괄호 <> 안에 "T" 를 넣어주면, "T"는 형식 매개변수가 됨

사용형식

```
한정자 반환_형식
메소드이름<형식_매개변수>(매개변수_목록)
{
    // ...
}
```

사용예제

```
void CopyArray<T>(T[] source, T[] target)
{
    for (int i = 0; i < source.Length; i++)
        target[i] = source[i];
}
```

일반화 메소드 호출 예제

```
int[] source = { 1, 2, 3, 4, 5 };
int[] target = new int[source.Length];

CopyArray<int>(source, target); // 형식 매개변수 T에 int를 대입

foreach(int element in target)
    Console.WriteLine(element);
```

일반화 메소드 예제 코드

```
static void CopyArray<T>(T[] source, T[] target)
{
    for (int i = 0; i < source.Length; i++)
        target[i] = source[i];
}

static void Main(string[] args)
{
    int[] source = { 1, 2, 3, 4, 5 };
    int[] target = new int[source.Length];

    CopyArray<int>(source, target);

    foreach(int element in target)
        Console.WriteLine(element);

    string[] source2 = { "하나", "둘", "셋", "넷", "다섯" };
    string[] target2 = new string[source2.Length];

    CopyArray<string>(source2, target2);

    foreach (string element in target2)
        Console.WriteLine(element);
}
```

출력 결과:

```
1
2
3
4
5
하나
둘
셋
넷
다섯
```

일반화 클래스

- 멤버의 데이터형식을 일반화하여 정의한 클래스
  - ✓ 클래스 이름 뒤에 꺽은 괄호 <> 안에 "T" 를 넣어주면, "T"는 형식 매개변수가 됨

```
Array_Generic<int> intArr = new Array_Generic<int>();
```

int 형식으로 사용될 때

```
class Array_Generic
{
    private int[] array;
    //...
    public int GetElement(int index) { return array[index]; }
}
```

일반화 클래스

```
class Array_Generic<T>
{
    private T[] array;
    //...
    public T GetElement(int index) { return array[index]; }
}
```

double 형식으로 사용될 때

```
Array_Generic<double> intArr = new Array_Generic<double>();
```

```
class Array_Generic
{
    private double[] array;
    //...
    public double GetElement(int index) { return array[index]; }
}
```

일반화 클래스 예제 코드

```
class MyList<T>
{
    private T[] array;

    public MyList()
    {
        array = new T[3];
    }

    public T this[int index]
    {
        get { return array[index]; }
        set
        {
            if(index >= array.Length)
            {
                Array.Resize<T>(ref array, index + 1);
                Console.WriteLine($"Array Resized : {array.Length}");
            }
            array[index] = value;
        }
    }

    public int Length { get { return array.Length; } }
}
```

출력 결과:

```
Array Resized : 4
Array Resized : 5
abc
def
ghi
jkl
mno

Array Resized : 4
Array Resized : 5
0
1
2
3
4
```

```
static void Main(string[] args)
{
    MyList<string> str_list = new MyList<string>();
    str_list[0] = "abc";
    str_list[1] = "def";
    str_list[2] = "ghi";
    str_list[3] = "jkl";
    str_list[4] = "mno";

    for (int i = 0; i < str_list.Length; i++)
        Console.WriteLine(str_list[i]);
    Console.WriteLine();

    MyList<int> int_list = new MyList<int>();
    int_list[0] = 0;
    int_list[1] = 1;
    int_list[2] = 2;
    int_list[3] = 3;
    int_list[4] = 4;

    for (int i = 0; i < int_list.Length; i++)
        Console.WriteLine(int_list[i]);
}
```

일반화 클래스 : 두개 이상의 형식 매개변수

- 일반화 프로그래밍에서 형식 매개변수는 두 개이상으로 사용 가능

```
class Wanted<T,U> // 두개의 형식매개변수 T, U
{
    public T Value1;
    public U Value2;

    public Wanted(T _value1, U _value2)
    {
        this.Value1 = _value1;
        this.Value2 = _value2;
    }
}

class MainApp
{
    static void Main(string[] args)
    {
        Wanted<int, string> wanted = new Wanted<int, string>(1234, "String");
        Console.WriteLine($"Value1: {wanted.Value1}, Value2: {wanted.Value2}");
    }
}
```

출력 결과:
Value1: 1234, Value2: String

형식 매개변수 제약시키기 : where 키워드

- 특정 조건을 갖춘 형식에만 대응하는 형식 매개변수를 사용하도록 제약
  - ✓ 메소드 또는 클래스에 where 키워드와 함께 제약조건을 나타냄

```
where 형식_매개변수 : 제약조건
```

제약	설명
where T : struct	T는 값 형식이어야 함
where T : class	T는 참조 형식이어야 함
where T : new()	T는 반드시 매개변수가 없는 생성자가 있어야함
where T : 기반_클래스_이름	T는 명시한 기반 클래스의 파생 클래스여야 함
where T : 인터페이스_이름	T는 명시한 인터페이스를 반드시 구현해야 함. 인터페이스_이름에는 여러 개의 인터페이스를 명시할 수도 있음
where T : U	T는 또 다른 형식 매개변수 U로부터 상속받는 클래스여야 함

형식 매개변수 제약시키기 사용 예제

- 형식 매개변수는 값 형식으로 제약

```
static void CopyArray<T>(T[] source, T[] target) where T : struct // T는 값형식으로 제약
{
    for (int i = 0; i < source.Length; i++)
        target[i] = source[i];
}

static void Main(string[] args)
{
    int[] source = { 1, 2, 3, 4, 5 };
    int[] target = new int[source.Length];

    CopyArray<int>(source, target);

    string[] source2 = { "하나", "둘", "셋", "넷", "다섯" };
    string[] target2 = new string[source2.Length];

    CopyArray<string>(source2, target2); // string 은 참조형식으로 에러발생
}
```

형식 매개변수 제약시키기 사용 예제

- 두 개 이상의 형식 매개변수 제약

```
class Test<T,U>
    where T : class
    where U : struct
{

}
```

T는 참조형식이여 함

U는 값 형식이여 함

```
class Test<T, U>
    where T : IComparable
    where U : IComparable, IDisposable
{

}
```

T는 IComparable 또는 IComparable 의 상속을 받은 것

U는 IComparable 과 IDisposable 의 상속을 받은 것

형식 매개변수 제약시키기 사용 예제

- 기본 생성자를 가진 어떤 클래스의 객체라도 생성

```
public static T CreateInstance<T>() where T : new()
{
    return new T();
}
```

- 특정 클래스로부터 상속 받은 형식 매개변수 U 사용

```
class BaseArray<U> where U : Base
{
    public U[] Array { get; set; }
    public BaseArray(int size)
    {
        Array = new U[size];
    }

    public void CopyArray<T>(T[] Source) where T : U // T는 상위 형식매개변수 U를 상속받도록 강제
    {
        Source.CopyTo(Array, 0);
    }
}
```

형식 매개변수 제약시키기 : 예제 코드

일반화 클래스 정의

```
class StructArray<T> where T : struct
{
    public T[] Array { get; set; }
    public StructArray(int size)
    {
        Array = new T[size];
    }
}

class RefArray<T> where T : class
{
    public T[] Array { get; set; }
    public RefArray(int size)
    {
        Array = new T[size];
    }
}
```

일반화 클래스 인스턴스 생성 및 값 할당

```
static void Main(string[] args)
{
    StructArray<int> a = new StructArray<int>(3);
    a.Array[0] = 0;
    a.Array[1] = 1;
    a.Array[2] = 2;

    RefArray<StructArray<double>> b = new RefArray<StructArray<double>>(3);
    b.Array[0] = new StructArray<double>(5);
    b.Array[1] = new StructArray<double>(10);
    b.Array[2] = new StructArray<double>(1005);
}
```

형식 매개변수 제약시키기 : 예제 코드

일반화 클래스 정의

```
class Base { }
class Derived : Base { }
class BaseArray<U> where U : Base
{
    public U[] Array { get; set; }
    public BaseArray(int size)
    {
        Array = new U[size];
    }

    public void CopyArray<T>(T[] Source) where T : U
    {
        Source.CopyTo(Array, 0);
    }
}
```

일반화 클래스 인스턴스 생성 및 값 할당

```
public static T CreateInstance<T>() where T : new()
{
    return new T();
}

static void Main(string[] args)
{
    BaseArray<Base> c = new BaseArray<Base>(3);
    c.Array[0] = new Base();
    c.Array[1] = new Derived();
    c.Array[2] = CreateInstance<Base>();

    BaseArray<Derived> d = new BaseArray<Derived>(3);
    d.Array[0] = new Derived();
    d.Array[1] = CreateInstance<Derived>();
    d.Array[2] = CreateInstance<Derived>();

    BaseArray<Derived> e = new BaseArray<Derived>(3);
    e.CopyArray<Derived>(d.Array);
}
```

일반화 프로그래밍

- 일반화 메소드/클래스
- 형식 매개변수의 제약
- 일반화 컬렉션

일반화 컬렉션

- 이전 수업에 배운 기존 컬렉션은 object 형식을 기반
  - ✓ ArrayList, Queue, Stack … etc
  - ✓ 모든 데이터형식을 다룰 수 있지만, 컬렉션 요소 접근 때마다 형식변환 수행
  - ✓ object 형식 기반이기 때문에 태생적인 성능문제를 가지고 있음

- 일반화 컬렉션은 object 형식 기반 컬렉션의 성능문제를 해결
  - ✓ 형식매개변수를 통해 컴파일할 때 컬렉션에 사용할 형식이 결정됨
  - ✓ List<T>, Queue<T>, Stack<T> … etc

일반화 컬렉션 : List <T> 예제 코드

- 비일반화 컬렉션 ArrayList와 같은 기능을 하고 사용법도 동일

```
static void Main(string[] args)
{
    List<int> list = new List<int>(); // int형 데이터를 관리하도록 정의
    for (int i = 0; i < 5; i++)
        list.Add(i); // i 값 추가

    foreach (int element in list)
        Console.Write($"{element} ");
    Console.WriteLine();

    list.RemoveAt(2); // index 2에 해당하는 값 삭제

    foreach (int element in list)
        Console.Write($"{element} ");
    Console.WriteLine();

    list.Insert(2, 2); // index 2에 2 값을 입력

    foreach (int element in list)
        Console.Write($"{element} ");
    Console.WriteLine();
}
```

출력 결과:
0 1 2 3 4
0 1 3 4
0 1 2 3 4

일반화 컬렉션 : Queue <T> 예제 코드

- 비일반화 컬렉션 Queue와 같은 기능을 하고 사용법도 동일

```
static void Main(string[] args)
{
    Queue<int> queue = new Queue<int>(); // int형 데이터를 관리하도록 정의

    queue.Enqueue(1);
    queue.Enqueue(2);
    queue.Enqueue(3);
    queue.Enqueue(4);
    queue.Enqueue(5);

    while (queue.Count > 0)
        Console.WriteLine(queue.Dequeue());
}
```

출력 결과:
1
2
3
4
5

일반화 컬렉션 : Stack <T> 예제 코드

- 비일반화 컬렉션 Stack와 같은 기능을 하고 사용법도 동일

```
class MainApp
{
    static void Main(string[] args)
    {
        Stack<int> stack = new Stack<int>(); // int형 데이터를 관리

        stack.Push(1);
        stack.Push(2);
        stack.Push(3);
        stack.Push(4);
        stack.Push(5);

        while (stack.Count > 0)
            Console.WriteLine(stack.Pop());
    }
}
```

출력 결과:
5
4
3
2
1

Thank you!